PIR

Theme Song: move

In this assignment, you will use somewhat homomorphic encryption to implement a private information retrieval scheme, which will allow clients to retrieve information from a server without the server learning what information they requested.

Due Date: *Friday, April 11*

Contents

1 Background Knowledge

In this assignment, you will implement a private information retrieval (PIR) scheme. This assignment leaves room for optimizations, which we leave to the reader to explore.

We recommend reading the first few pages of Communication–Computation Trade-offs in PIR.

1.1 Homomorphic Encryption

We have already encountered additively homomorphic encryption in the Vote project. In this project, we will utilize an encryption scheme that supports both homomorphic addition and homomorphic multiplication. In particular, consider an encryption scheme with additional homomorphic evaluation algorithms HomAdd and HomMul such that for any two messages m_0, m_1, we have both that $\mathsf{HomAdd}(\mathsf{Enc}(m_0), \mathsf{Enc}(m_1)) = \mathsf{Enc}(m_0 + m_1)$ and that $\mathsf{HomMul}(\mathsf{Enc}(m_0), \mathsf{Enc}(m_1)) = \mathsf{Enc}(m_0 \cdot m_1)$. In other words, we can construct a ciphertext for $m_0 + m_1$ or $m_0 \cdot m_1$ using ciphertexts for m_0 and m_1 individually.

Achieving fully homomorphic encryption which supports homomorphic evaluation for all polynomial-sized circuits is fairly inefficient due to the expensive bootstrapping step. Nevertheless, for most practical applications such as private information retrieval (PIR), it suffices to have a slightly weaker primitive known as somewhat homomorphic encryption (SWHE). SWHE also supports both homomorphic addition and homomorphic multiplication, but allows for only a bounded number of homomorphic operations.

1.2 Private Information Retrieval

We turn our attention to the problem of private information retrieval (PIR). In PIR, we have a server and a client, where the server holds a database of n plaintexts $D = \{p_1, \dots, p_n\}$ and the client wants to retrieve p_k for some $k \in [n]$. We would like to allow the client to retrieve p_k without the server learning k. This is distinct from OT as the client is allowed to learn as many values as she wants. There is a trivial solution in which the server simply sends the entire database to the client; however, this requires communication complexity of $O(n)$, and our goal in PIR is to achieve lower (sublinear) communication complexity.

We can use somewhat homomorphic encryption to achieve a PIR scheme. If the client wants to retrieve p_k, we can send a selection vector $s = (s_1, \dots, s_n)$ where s_k is an encryption of 1 and every other s_i is an encryption of 0. Now, the server can homomorphically compute $p_i' = p_i \cdot s_i$ and then sum up all values of p_i' to retrieve an encryption of just the selected value. The communication complexity from the server to the client is constant (namely a single ciphertext), which is a huge improvement compared to sending the entire database. However, the communication cost from the client to the server is still $O(n)$.

By organizing the database into a square of side length $\sqrt{n}$ and sending a selection vector for each dimension (x, y), the client only needs to send $2 \cdot \sqrt{n}$ ciphertexts. However, we end up incurring more ciphertext multiplications on the server side. This tradeoff between communication and computation is hard to strike perfectly. We can generalize this approach to organize our database into a hypercube of side length $\sqrt[d]{n}$, and then sending $d \cdot \sqrt[d]{n}$ ciphertexts from the client to the server. In this project we will implement the generic protocol to work with any dimension.

2 Assignment Specification

Please note: you may NOT change any of the function headers defined in the stencil. Doing so will break the autograder; if you don't understand a function header, please ask us what it means and we'll be happy to clarify.

2.1 Functionality

You will primarily need to edit `src/pkg/cloud.cxx` and `src/pkg/agent.cxx`. The following is an overview of relevant files:

1. `src/cmd/cloud.cxx` is the main entrypoint for the `pir_cloud` binary. It calls the `Cloud` class.
2. `src/cmd/agent.cxx` is the main entrypoint for the `pir_agent` binary. It calls the `Agent` class.
3. `src/drivers/hypercube_driver.cxx` contains a driver for a hypercube data store.
4. `src/pkg/cloud.cxx` implements the `Cloud` class.
5. `src/pkg/agent.cxx` implements the `Agent` class.

2.1.1 Homomorphic Encryption and Decryption

Implement homomorphic encryption and decryption using SEAL. Once you do so, you'll be able to encrypt selection vectors and decrypt the final result.

Application functions:

- `AgentClient::DoRetrieve(...)`
- `CloudClient::HandleSend(...)`

2.2 Support Code

Read the support code header files before coding so you have a sense of what functionality we provide. This isn't a networking class, nor is it a software engineering class, so we try to abstract away as many of these details as we can so you can focus on the cryptography.

The following is an overview of the functionality that each support code file provides.

1. `src/drivers/hypercube_driver.cxx` contains a hypercube datastore.

2. `src-shared/messages.cxx` contains a new type of message, `SerializableWithContext`. The difference from `Serializable` messages is that these must have a `SEALContext` passed in during a call to deserialize into a struct. The query and response messages are of this type.

3. Everything else from prior assignments is unchanged.

2.3 Libraries: SEAL

For this assignment, we'll be using a new library that implement a SWHE protocol, BFV. The code comments and the SEAL tutorials will guide you through using the library. You may find the following repository useful during this assignment: SEAL. Be sure to check out the examples, they will be immensely helpful!

3 Getting Started

To get started, get your stencil repository here and clone it into the `devenv/home` folder. From here you can access the code from both your computer and from the Docker container.

Before building the project, you will need to install the SEAL library. Do so by running the following commands in order:

```
git clone https://github.com/microsoft/SEAL.git
cd ~/SEAL
cmake -S . -B build
cmake --build build
sudo cmake --install build
```

3.1 Running

To build the project, `cd` into the `build` folder and run `cmake ..`. This will generate a set of Makefiles building the whole project. From here, you can run `make` to generate a binary you can run, and you can run `make check` to run any tests you write in the `test` folder.

3.2 Testing

You may write tests in any of the `test/**.cxx` files in the Doctest format. We provide `test/test_provided.cxx`, feel free to write more tests in this file directly. If you want to add any new test *files*, make sure to add the file to the `cmake` variable, `TESTFILES`, on line 7 of `test/CMakeLists.txt` so that `cmake` can pick up on the new files. Examples have been included in the assignment stencil. To build and run the tests, run `make check` in the `build` directory. If you'd like to see if your code can interoperate with our code (which is what it will be tested against), feel free to download our binaries here.

Note that testing locally using `make check` may result in a `Killed` message – this doesn't mean your code isn't working, it just means that the library ran out of memory. Try running it with smaller parameters ($d \leq 3, s \leq 10$).